Lecture 9: Division

Fall 2023
Jason Tang
Topics

- Dividing unsigned numbers
- Hardware designs for dividers
- Division in C
Unsigned Integer Division, Longhand

• Dividing binary values is like dividing decimals by hand:

\[
\begin{array}{c|c}
\text{Dividend} & \text{Quotient} \\
\hline
1001010 & 9_{10} \\
- 1000 & \\
\hline
1010 & 74_{10} \\
- 1000 & \\
\hline
10 & \text{Remainder or modulo result} \end{array}
\]

= \frac{1001010}{1000} = 74_{10} \text{ (Dividend)}

• Dividend = Quotient \times \text{Divisor} + \text{Remainder}

• Each bit in quotient means that divisor could be subtracted from partial dividend: 0 \rightarrow \text{smaller than divisor}, 1 \rightarrow \text{greater than or equal to divisor}
Unsigned Shift-Subtract Divider (version 1)

- Store $n$-bit divisor in a register twice its size, towards MSB
- For each step, shift divisor right and quotient left
- Initialize $2n$-bit remainder register to dividend, towards LSB
- **Control** decides when to shift and when to write new value into remainder register
Example of Shift-Subtract Divider (version 1)

• Dividing two $n$-bit numbers needs $n+1$ steps to generate $n$-bit quotient and remainder

  • Subtract divisor from remainder

  • If result is non-negative, keep new remainder and append 1 to quotient,

  • Else restore remainder, and append 0 to quotient

  • Logical shift divisor right, then shift quotient left

  • Repeat $n+1$ times

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Operations</th>
<th>Quotient</th>
<th>Remainder</th>
<th>Divisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial</td>
<td>0000</td>
<td>0000 0111</td>
<td>0010 0000</td>
</tr>
<tr>
<td>1a</td>
<td>Subtract</td>
<td>0000</td>
<td>1110 0111</td>
<td>0010 0000</td>
</tr>
<tr>
<td>1b</td>
<td>Restore remainder</td>
<td>0000</td>
<td>0000 0111</td>
<td>0010 0000</td>
</tr>
<tr>
<td>1c</td>
<td>Logical shift divisor right</td>
<td>0000</td>
<td>0000 0111</td>
<td>0001 0000</td>
</tr>
<tr>
<td>1d</td>
<td>Shift quotient left</td>
<td>0000</td>
<td>0000 0111</td>
<td>0001 0000</td>
</tr>
<tr>
<td>after 2</td>
<td>Sub, restore, shift D, shift Q</td>
<td>0000</td>
<td>0000 0111</td>
<td>0000 1000</td>
</tr>
<tr>
<td>after 3</td>
<td>Sub, restore, shift D, shift Q</td>
<td>0000</td>
<td>0000 0111</td>
<td>0000 0100</td>
</tr>
<tr>
<td>after 4</td>
<td>Sub, keep, shift D, shift Q</td>
<td>0001</td>
<td>0000 0011</td>
<td>0000 0010</td>
</tr>
<tr>
<td>after 5</td>
<td>Sub, keep, shift D, shift Q</td>
<td>0011</td>
<td>0000 0001</td>
<td>0000 0001</td>
</tr>
</tbody>
</table>
Unsigned Shift-Subtract Divider (version 2)

- Because half of divisor register is filled with zeroes, $2n$-bit ALU is wasteful.
- Use only $n$-bit divisor, $n$-bit ALU; keep $2n$-bit remainder.
- Result of ALU is written to upper half of remainder register.
- After each step, control shifts left the remainder register.

![Diagram of Unsigned Shift-Subtract Divider (version 2)]
Example of Shift-Subtract Divider (version 2)

- Subtract from upper half of remainder
- If test subtraction is non-negative, keep result in upper half of remainder
- Because iteration 1 cannot produce a 1 in quotient, first shift remainder and then subtract to save an iteration
- Repeat \( n \) (instead of \( n+1 \)) times

### Example: \( 7_{10} \div 2_{10}, 4\text{-bits} \)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Operations</th>
<th>Quotient</th>
<th>Remainder</th>
<th>Divisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial</td>
<td>0000</td>
<td>0000 0111</td>
<td>0010</td>
</tr>
<tr>
<td>1a</td>
<td>Shift remainder left</td>
<td>0000</td>
<td>0000 1110</td>
<td>0010</td>
</tr>
<tr>
<td>1b</td>
<td>Subtract</td>
<td>0000</td>
<td>1110 1110</td>
<td>0010</td>
</tr>
<tr>
<td>1c</td>
<td>Restore</td>
<td>0000</td>
<td>0000 1110</td>
<td>0010</td>
</tr>
<tr>
<td>1d</td>
<td>Shift quotient left</td>
<td>0000</td>
<td>0000 1110</td>
<td>0010</td>
</tr>
<tr>
<td>after 2</td>
<td>Shift R, sub, restore, shift Q</td>
<td>0000</td>
<td>0001 1100</td>
<td>0010</td>
</tr>
<tr>
<td>after 3</td>
<td>Shift R, sub, keep, shift Q</td>
<td>0001</td>
<td>0001 1000</td>
<td>0010</td>
</tr>
<tr>
<td>after 4</td>
<td>Shift R, sub, keep, shift Q</td>
<td>0011</td>
<td>0001 0000</td>
<td>0010</td>
</tr>
</tbody>
</table>
Unsigned Shift-Subtract Divider (version 3)

- At startup, top half of remainder register will be shifted away and is unused

- Combine quotient and remainder registers, with remainder in top half of register and quotient in bottom half

- ALU results still written to upper half of combined register
Example of Shift-Subtract Divider (version 3)

- After \( n \) iterations, upper half of combined register holds the remainder, bottom half holds quotient

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Operations</th>
<th>Remainder / Quotient</th>
<th>Divisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial</td>
<td>0000 0111</td>
<td>0010</td>
</tr>
<tr>
<td>1a</td>
<td>Shift R/Q left</td>
<td>0000 1110</td>
<td>0010</td>
</tr>
<tr>
<td>1b</td>
<td>Subtract</td>
<td>1110 1110</td>
<td>0010</td>
</tr>
<tr>
<td>1c</td>
<td>Restore</td>
<td>0000 1110</td>
<td>0010</td>
</tr>
<tr>
<td>after 2</td>
<td>Shift R/Q, sub, restore</td>
<td>0001 1100</td>
<td>0010</td>
</tr>
<tr>
<td>after 3</td>
<td>Shift R/Q, sub, keep</td>
<td>0001 1001</td>
<td>0010</td>
</tr>
<tr>
<td>after 4</td>
<td>Shift R/Q, sub, keep</td>
<td>0001 0011</td>
<td>0010</td>
</tr>
</tbody>
</table>

Example: \( 7_{10} \div 2_{10}, 4\)-bits
Multiplication versus Division

- Hardware multipliers can be optimized to run in only a few cycles
  - Parallel adders; algorithm tricks like Booth, Karatsuba, and Toom-Cook

- Hardware dividers all rely upon some kind of iterative approach

- For a typical ARMv8-A implementation:

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Number of Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer add or subtract</td>
<td>1</td>
</tr>
<tr>
<td>Integer multiply</td>
<td>3</td>
</tr>
<tr>
<td>Multiply-accumulate</td>
<td>4</td>
</tr>
<tr>
<td>Integer division</td>
<td>12</td>
</tr>
</tbody>
</table>

https://www.anandtech.com/show/12785/arm-cortex-a76-cpu-unveiled-7nm-powerhouse/3
Optimizing Dividers

• Instead of subtracting the divisor, add the divisor’s inverse

• Instead of keeping/restoring the combined register, use a mux to select which bits to write back to the register

  • If MSB out of ALU is 0, then result is non-negative, which means to keep change

  • Else if MSB is 1, then result is negative, which is a restore

• Some implementations will check if the divisor is zero

  • Will raise an interrupt if a dividing by zero
Unsigned Shift-Subtract Divider Implementation
Signed Division

- Simplest approach is to remember signs, make both dividend and divisor positive, and then apply correct signs afterwards

- Quotient’s sign is negative when dividend and divisor’s sign differ
  
  \[ Q_{\text{sign}} = \text{Dividend}_{\text{sign}} \oplus \text{Divisor}_{\text{sign}} \]

- Remainder’s sign is same as dividend’s sign

- Otherwise, \(-7 \div 2\) could be \(-4\), remainder \(+1\)

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>(+3), remainder (+1)</td>
</tr>
<tr>
<td>7</td>
<td>(-2)</td>
<td>(-3), remainder (+1)</td>
</tr>
<tr>
<td>(-7)</td>
<td>2</td>
<td>(-3), remainder (-1)</td>
</tr>
<tr>
<td>(-7)</td>
<td>(-2)</td>
<td>(+3), remainder (-1)</td>
</tr>
</tbody>
</table>
Division in C

• Result of `/` operator has the same sign as the dividend

• In C99, `/` rounds towards 0 (relevant with a negative operand)

• The `%` operator is the modulo operator

• By definition, \((a / b) \times b + (a \% b) = a\)

```c
void f(int a, int b) {
    printf("%d / %d = %d, mod %d\n", 
           a, b, (a/b), (a%b));
}
int main(void) {
    f(7, 2);    f(7, -2);
    f(-7, 2);   f(-7, -2);
    return 0;
}
```

7 / 2 = 3, mod 1
7 / -2 = -3, mod 1
-7 / 2 = -3, mod -1
-7 / -2 = 3, mod -1

https://en.cppreference.com/w/c/language/operator_arithmetic
Division Instructions

- Two different types of division:

<table>
<thead>
<tr>
<th>Division Type</th>
<th>ARMv8-A</th>
<th>PowerPC</th>
<th>x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed Division</td>
<td>sdiv</td>
<td>divw</td>
<td>idiv</td>
</tr>
<tr>
<td>Unsigned Division</td>
<td>udiv</td>
<td>divwu</td>
<td>div</td>
</tr>
</tbody>
</table>

- For ARMv8-A and PowerPC, no hardware check for divide by zero

- For ARMv8-A and PowerPC, no builtin way to get remainder / modulo
```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    signed long s1 = strtol(argv[1], NULL, 0);
    signed long s2 = strtol(argv[2], NULL, 0);
    unsigned long u1 = strtoul(argv[1], NULL, 0);
    unsigned long u2 = strtoul(argv[2], NULL, 0);

    signed long squot = s1 / s2;
    signed long smod = s1 % s2;
    unsigned long uquot = u1 / u2;
    unsigned long umod = u1 % u2;
    printf("  signed a / b = %ld, a %% b = %ld\n", squot, smod);
    printf("unsigned a / b = %lu, a %% b = %lu\n", uquot, umod);
    return 0;
}
```
x86-64 Division

- On x86-64, 128-bit signed division is performed by the `idivq` instruction:
  - `rdx` holds upper 64 bits of dividend
  - `rax` holds lower 64 bits of dividend
  - Operand to `idivq` is the divisor
  - Afterwards, quotient will be in `rax`, modulo in `rdx`
x86-64 Division

\[
\text{signed long } \text{squot} = \text{s1} / \text{s2}; \\
400649: \quad 48 \text{ 8b 45 c0} \quad \text{mov} \quad -0x40(%rbp),%rax \\
40064d: \quad 48 \text{ 99} \quad \text{cqto} \\
40064f: \quad 48 \text{ f7 7d c8} \quad \text{idivq} \quad -0x38(%rbp) \\
400653: \quad 48 \text{ 89 45 e0} \quad \text{mov} \quad %rax,-0x20(%rbp) \\
\]

\[
\text{signed long } \text{smod} = \text{s1} \% \text{s2}; \\
400657: \quad 48 \text{ 8b 45 c0} \quad \text{mov} \quad -0x40(%rbp),%rax \\
40065b: \quad 48 \text{ 99} \quad \text{cqto} \\
40065d: \quad 48 \text{ f7 7d c8} \quad \text{idivq} \quad -0x38(%rbp) \\
400661: \quad 48 \text{ 89 55 e8} \quad \text{mov} \quad %rdx,-0x18(%rbp) \\
\]

\[
\text{unsigned long } \text{uquot} = \text{u1} / \text{u2}; \\
400665: \quad 48 \text{ 8b 45 d0} \quad \text{mov} \quad -0x30(%rbp),%rax \\
400669: \quad \text{ba 00 00 00 00} \quad \text{mov} \quad $0x0,%edx \\
40066e: \quad 48 \text{ f7 75 d8} \quad \text{divq} \quad -0x28(%rbp) \\
400672: \quad 48 \text{ 89 45 f0} \quad \text{mov} \quad %rax,-0x10(%rbp) \\
\]

\[
\text{unsigned long } \text{umod} = \text{u1} \% \text{u2}; \\
400676: \quad 48 \text{ 8b 45 d0} \quad \text{mov} \quad -0x30(%rbp),%rax \\
40067a: \quad \text{ba 00 00 00 00} \quad \text{mov} \quad $0x0,%edx \\
40067f: \quad 48 \text{ f7 75 d8} \quad \text{divq} \quad -0x28(%rbp) \\
400683: \quad 48 \text{ 89 55 f8} \quad \text{mov} \quad %rdx,-0x8(%rbp) \\
\]
Intel Division Implementation

• Original implementation used a shift-subtract divider, about half the speed of shift-adder multiplication

• Newer chips use a SRT table to look up a quotient digit, based upon dividend and divisor values

• Intel Pentium chips had a famous hardware bug due to incorrectly encoded SRT table

ARMv8-A Division

• On ARMv8-A, 64-bit signed division is performed by the `sdiv` instruction

  • Divisor, dividend, and quotient registers are all specified by instruction operands

• Upon a divide-by-zero, 0 is written to quotient register, with no other indication that any problem occurred

• To get the modulo, multiply the quotient by dividend, and then subtract that from the divisor
signed long squot = s1 / s2;
4006e0:       f94013a1        ldr     x1, [x29,#32]
4006e4:       f94017a0        ldr     x0, [x29,#40]
4006e8:       9ac00c20        sdiv    x0, x1, x0
4006ec:       f90023a0        str     x0, [x29,#64]

signed long smod = s1 % s2;
4006f0:       f94013a0        ldr     x0, [x29,#32]
4006f4:       f94017a1        ldr     x1, [x29,#40]
4006f8:       9ac10c02        sdiv    x2, x0, x1
4006fc:       f94017a1        ldr     x1, [x29,#40]
400700:       9b017c41        mul     x1, x2, x1
400704:       cb010000        sub     x0, x0, x1
400708:       f90027a0        str     x0, [x29,#72]

unsigned long uquot = u1 / u2;
40070c:       f9401ba1        ldr     x1, [x29,#48]
400710:       f9401fa0        ldr     x0, [x29,#56]
400714:       9ac00820        udiv    x0, x1, x0
400718:       f9002ba0        str     x0, [x29,#80]